**Building a distributed system requires
a methodical approach to requirements.**

BY MARK CAVAGE

There Is No Getting Around It: You Are Building a Distributed System

DISTRIBUTED SYSTEMS ARE difficult to understand, design, build, and operate. They introduce exponentially more variables into a design than a single machine does, making the root cause of an application problem much harder to discover. It should be said that if an application does not have meaningful service-level

agreements (SLAs) and can tolerate extended downtime and/or performance degradation, then the barrier to entry is greatly reduced. Most modern applications, however, have an expectation of resiliency from their users, and SLAs are typically measured by "the number of nines" (for example, 99.9 or 99.99% availability per month). Each additional nine becomes more difficult to achieve.

To complicate matters further, it is extremely common that distribut-

ed failures will manifest as intermittent errors or decreased performance (commonly known as brownouts). These failure modes are much more time consuming to diagnose than a complete failure. For example, Joyent operates several distributed systems as part of its cloud-computing infrastructure. In one such system—a highly available, distributed key/value store—Joyent recently experienced transient application timeouts. For most users the system oper-

ated normally and responded within the bounds of its latency SLA. However, 5%–10% of requests would routinely exceed a predefined application timeout. The failures were not reproducible in development or test environments, and they would often "go away" for minutes to hours at a time. Troubleshooting this problem to root cause required extensive system analysis of the data-storage API (node.js), an RDBMS (relational database management system) used internally by the system (PostgreSQL), the operating system, and the end-user application reliant on the key/value system. Ultimately, the root problem was in application semantics that caused excessive locking, but determining root cause required considerable data gathering and correlation, and consumed many working hours of time among engineers with differing areas of expertise.

The entire preamble is fairly common knowledge to engineers working on distributed systems, and the example application degradation is typical of a problem that arises when operating a large system. Current computing trends, however, are bringing many more organizations and applications into the distributed-computing realm than was the case only a few years ago. There are many reasons why an organization would need to build a distributed system, but here are two examples:

▸ The demands of a consumer website/API or multitenant enterprise application simply exceed the computing capacity of any one machine.

▸ An enterprise moves an existing application, such as a three-tier system, onto a cloud service provider in order to save on hardware/data-center costs.

The first example is typical of almost any Internet-facing company today. This has been well discussed for many years and is certainly a hotbed for distributed-systems research and innovation, and yet it is not a solved problem. Any application going live still needs to assess its own requirements and desired SLAs, and design accordingly to suit its system; there is no blueprint. The usage patterns of any new successful applications will likely be different from those that came before, requiring, at best, cor-

Too many applications are being built by stringing together a distributed database system with some form of application and hoping for the best. Instead—as with all forms of engineering—a methodical, data-driven approach is needed.

rect application of known scaling techniques and, at worst, completely new solutions to their needs.

The second example is increasingly common, given current trends to move existing business applications into the cloud to save on data-center costs, development time, etc. In many cases, these existing applications have been running on single-purpose systems in isolated or low-utilization environments. Simply dropping them into an environment that is often saturated induces failure more often than these applications have ever seen or were designed for. In these cases, the dirty secret is that applications must be rewritten when moving to cloud environments; to design around the environment in which an application will be running requires that it be built as a distributed system.

These two examples are at opposite extremes in terms of the reasons why they require a distributed solution, yet they force system designers to address the same problems. Making matters worse, most real-world distributed systems that are successful are built largely by practitioners who have gone through extensive academic tutelage or simply "come up hard," as it were. While much of the discussion, literature, and presentations focus on the en vogue technologies such as Paxos,[3] Dynamo,[5] or MapReduce[4]—all of which indeed merit such focus—the reality of building a large distributed system is that many "traditional" problems are left unsolved by off-the-shelf solutions, whether commercial or open source.

This article takes a brief walk through the reality of building a real-world distributed system. The intent is neither to provide a toolkit, or any specific solutions to any specific problems, nor to dissuade new builders from constructing and operating distributed systems. Instead, the goal is simply to highlight the realities and look at the basics required to build one. This is motivated by a trend toward building such systems on "hope and prayer." Too many applications are being built by stringing together a distributed database system with some form of application and hoping for the best. Instead—as with all forms of engineering—a methodical, data-driven

approach is needed, and the goal of this article is to provide useful tips on using such a methodology.

While it is important to disambiguate "the cloud" from a distributed system, for practical reasons most new distributed systems are being built in a cloud—usually a public one—and so the rest of this article assumes that environment. Furthermore, this article focuses specifically on online Web services (what once upon a time was referred to as OLTP—online transaction processing—applications), as opposed to batch processing, information retrieval, among others, which belong in an entirely separate problem domain. Finally, this article leverages a hypothetical—but not purely imagined—application in order to provide concrete examples and guidance at each stage of the development life cycle.

Architecting a Distributed System

One of the most common terms that applies to a distributed-system architecture is service-oriented architecture (SOA). While SOA may conjure (to some readers) unpleasant flashbacks of CORBA brokers and WS-* standards, the core ideas of loosely coupled services—each serving a small function and working independent from one another—are solid, and they are the foundation of a resilient distributed system. Drawing parallels to a previous generation, services are the new process; they are the correct abstraction level for any discrete functionality in a system.

The first step in constructing a service-oriented architecture is to identify each of the functions that comprise the overall business goals of the application and map these into discrete services that have independent fault boundaries, scaling domains, and data workload. For each of the services identified, you must consider the following items:

▸ *Geographies.* Will this system be global, or will it run in "silos" per region?

▸ *Data segregation.* Will this system offer a single- or multi-tenancy model?

▸ *SLAs.* Availability, latency, throughput, consistency, and durability guarantees must all be defined.

▸ *Security.* IAAA (identity, authentication, authorization, and audit), data confidentiality, and privacy must all be considered.

▸ *Usage tracking.* Understanding usage of the system is necessary for at minimum day-to-day operations of the system, as well as capacity planning.[1] It may also be used to perform billing for use of the system and/or governance (quota/rate limits).

▸ *Deployment and configuration management.* How will updates to the system be deployed?

This is not a comprehensive list of what it takes to operate a distributed system (and certainly not a list of requirements for running a business based on a distributed system, which brings a lot more complexity); rather it is a list of items to be addressed as you define the services in a distributed system. Additionally, note that all the "standard" software-engineering problems, such as versioning, upgrades, and support, must all still be accounted for, but here the focus is purely on the aspects that are core (but not necessarily unique) to a given service in a distributed system.

Consider the following hypothetical application: a Web service that simply resizes images. On the surface this appears to be a trivial problem, as image resizing should be a stateless affair, and simply a matter of asking an existing operating system and library or command to perform an image manipulation. However, even this very basic service has a plethora of pitfalls if your desire is to operate it at any scale. While the business value of such a service may be dubious, it will suit our purposes here of modeling what a distributed system around this utility would look like, as there is enough complexity to warrant quite a few moving parts—and it only gets harder the more complex the service.

Example System: Image-Resizing Service

Let's define the characteristics of the example Web service from a business point of view. Since the service will simply resize images, the system will need to ingest images, convert them, and make them available to the customer as quickly as possible—with a response time that is acceptable to an interactive user agent (that is, Web browser). Given the resizing is to be done in real time, and to simplify this article, let's assume the images are immediately downloaded and their long-term storage is not within the confines of this system. Customers will need to be able to securely identify themselves (the scheme by which they do so is outside the scope of this article, but we can assume they have some way of mapping credentials to identity). The service should offer a usage-based pricing model, so it needs to keep track of several dimensions of each image (for each request), such as the size of image and the amount of CPU time required to convert it. Finally, let's assume each data center where the system operates is isolated from all others, with no cross-data-center communication.

In terms of adoption, let's just throw a dart at the wall and say the number of users is to be capped at 100,000 and the number of requests per second is to be 10,000 in a single region (but this number will be available in each region). The business needs the system to provide 99.9% availability in a month, and the 99th percentile of latency should be less than 500ms for images less than one megabyte in size.

While this is a contrived example with an extremely short list of requirements—indeed, a real-world system would have substantially better-qualified definitions—we can already identify several distinct functions of this system and map them into discrete services.

Break Down Into Services

As stated earlier, the initial step is to decompose the "business system" into discrete services. At first glance, it may seem an image-resizing service needs only a few machines and some low-bar way to distribute resizes among them. Before going any further, let's do some back-of-the-envelope math as to whether this holds true or not.

Recall the goal is to support 10,000 resizes per second. For the purposes of this article, let's limit the range of input formats and define an average input size to be 256KB per image, so the system can process 10 conversions per second per CPU core. Given modern CPU architectures, we can select a system that has 32 cores (and enough

DRAM/IOPS capacity so there is no worry about hitting other boundaries for the purpose of this example). Obviously such a system can support 320 conversions per second, so to support the goals of 10,000 images per second, the business needs a fleet of at least 32 image-processing servers to meet that number alone. To maintain 20% headroom for surges, an extra seven machines is required. To point at a nice even number (and since this is notepad math anyway), the ballpark figure would be 40 systems to meet these goals. As the business grows, or if the average conversion time goes up, it will need more systems, so assume that 40 is a lower bound. The point is, this image service needs enough systems that it requires a real distributed stack to run it.

Now that we have confirmed a distributed system is necessary, the high-level image requirements can be mapped into discrete services. I will first propose an architecture based on my experience building distributed systems, though there are many alternative ways this could be done. The justification for the choices will follow this initial outlay; in reality, arriving at such an architecture would be much more of an iterative process.

The diagram in the accompanying figure breaks down the system into a few coarse services:

▸ A customer-facing API (such as REST, or representational state transfer, servers).

▸ A distributed message queue for sending requests to compute servers.

▸ An authoritative identity-management system, coupled to a high-performance authentication "cache."

▸ A usage aggregation fleet for operations, billing, and so on.

There are, of course, other ways of decomposing the system, but the above example is a good starting point; in particular, it provides a separation of concerns for failure modes, scaling, and consistency, which we will elaborate on as we iterate through the components.

Image API Details

Starting from the outside in, let's examine the image API tier in detail. First, a key point must be made: the API servers should be stateless. You should strive to make as many pieces in a distributed system as stateless as possible. State is always where the most difficult problems in all aspects arise, so if you can avoid keeping state, you should.

▸ *Geographies.* As the requirements are to operate in a single region, the API server(s) offers access to the system only for a single region.

▸ *Data segregation.* API servers are stateless, but it is worth pointing out that to deliver 10,000 resizes per second across a large set of customers, the system must be multitenant. The variance at any given time across customers means we would be unable to build any dedicated systems.

▸ *SLA availability.* Availability is the worst number either the software/systems or the data center/networking can offer. Since the business wants 99.9% availability from the system, let's set a goal for this tier at 99.99% availability in a given month, which allows for 4.5 minutes of downtime per month. This number is measured from the edge of the data center, since the system cannot control client network connectivity.

▸ *SLA latency.* The latency of any request in the API server is the sum of the latencies to each of the dependent systems. As such, it must be as low as possible: single-digit milliseconds to (small) tens of milliseconds at the 99th percentile.

▸ *SLA throughput.* As defined earlier, the system must support 10,000 requests per second. Given a non-native implementation, we should be able to support 5,000 requests per second on a given server (this number is drawn purely from practical experience). Keeping with a goal of 20% headroom, the system needs three servers to handle this traffic.

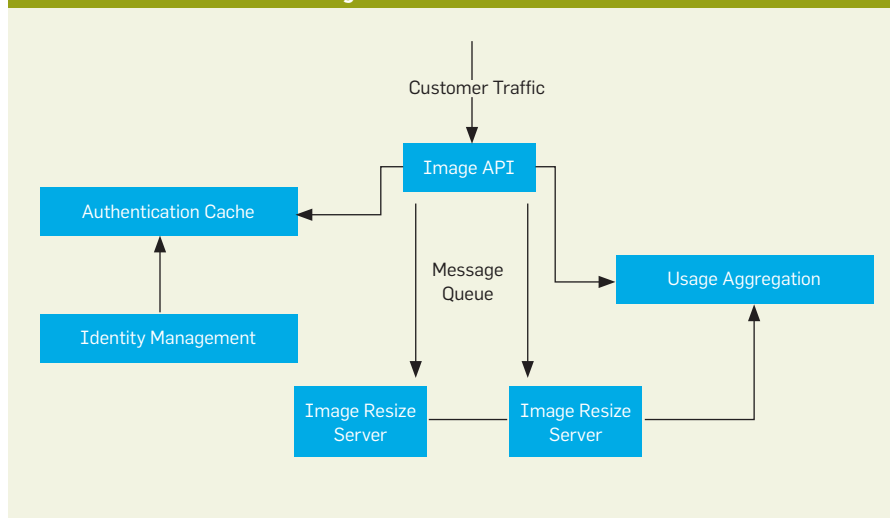▸ *SLA consistency and durability.* The API tier is stateless.

▸ *IAAA.* Each customer needs his or her own identity/credentials. Users must not be able to access each other's input or output images (recall we are not offering long-term storage, where sharing would be meaningful). A record of each resize request must be saved.

▸ *Usage tracking.* We want to "meter" all requests on a per-customer basis, along with the request parameters—when implementing the API, we need to ensure the system captures all dimensions about the request to be passed along.

▸ *Deployment and configuration management.* A goal for system operations is to manage all deployments and configuration of the system in one manner. We can table this for now and touch on it later on.

We have established the need for more than one API server, so load balancing of traffic must be managed across all API servers. Since there is ample prior art[10] and known solutions to the problem, for this particular aspect I will be brief and simply point out that it must be addressed. Solutions typically involve some combination of round-robin DNS (Domain System) records to public IP addresses, where these IP addresses are greater

The distributed services of an image resize service.

Customer Traffic

Image API

Authentication Cache

Identity Management

Message Queue

Usage Aggregation

Image Resize Server

Image Resize Server

than one load balancer resolved by protocols such as CARP (Common Address Redundancy Protocol) or similar solutions involving IP address takeovers at L2/L3 (layer 2/layer 3).

Finally, it is worth pointing out the API tier will certainly need a custom implementation. The semantics of this are so tied to the business that in the build-or-buy trade-off, there is really no choice but to build. Load balancing, however, offers many open source and commercial choices, so this is a good place to start. Specialized aspects of the API may require us to build our own, but it would not be prudent to start by assuming so.

Messaging

There is a discrepancy in scale between the API servers and the image-processing servers, thus requiring a scheduling mechanism to deliver images to available transcoding servers. There are many ways to achieve this, but given the individual systems can schedule their own cores, a relatively obvious starting point is to use a distributed message queue. A message queue could "stage" images locally on the API server that took the user request and push a ready message onto the queue. This has the nice property of providing a first-pass scheduling mechanism—FIFO—and allows graceful latency degradation of user resize times when the system is too busy. It should be stated again, however, that as is true of all components in a distributed system, message queues are not without trade-offs. In particular, message queues typically are often viewed as both highly available and consistent, but as the CAP theorem (meaning consistency, availabilit,y and partition tolerance) says, we cannot have our cake and eat it too.

Generally speaking, message queues prioritize consistency over availability; most open source and commercial message queues are going to prioritize consistency and run in some nature of active/standby mode, where the standby mode gets a (hopefully) synchronous copy of messages that are pushed and popped from the queue. While this is likely the right common choice, it does mean the example system must automate downtime, as promotion of a standby and

It is worth pointing out the API tier will certainly need a custom implementation. The semantics of this are so tied to the business that in the build-or-buy trade-off, there is really no choice but to build.

updating replication topologies are nontrivial. Before discussing what that solution would look like, let's run through the requirements again:

▶ *Geographies.* This is not applicable. We will run a message queue per deployment.

▶ *Data segregation.* Multitenant—there is one logical message queue per deployment.

▶ *SLA availability.* The system needs to provide 99.9% availability, and since we are assuming we will not be writing a new message queue from scratch, we need to automate failover to minimize this number as much as possible.

▶ *SLA latency.* Latency in a non-queued case (obviously if the consumer is reading slowly, latency goes up) needs to be as small as possible. The goal is sub-millisecond latency on average, and certainly single-digit millisecond latency for the 99th percentile of requests. Luckily, most message queues are capable of providing this, given they are in-memory.

▶ *SLA throughput.* Because we are using the messaging queue bidirectionally, the system needs to support 20,000 requests per second (24,000 with 20% headroom). The number of systems needed to support this is dependent on the choice of message-queue implementation.

▶ *SLA consistency and durability.* As previously stated, the queue must guarantee FIFO semantics, so we must have a strong consistency guarantee.

▶ *IAAA.* This is not applicable.

▶ *Usage tracking.* While we are interested in operational metrics, this is not applicable.

▶ *Deployment and configuration management.* Failovers must be automated to meet the availability SLAs.

Automating Failover

To automate failovers, the system needs an extra mechanism in place to perform leader election. While there is much literature on the subject of leader election, and many algorithms have come in over the years, we would like to have a distributed consensus protocol (for example, Paxos) in place. Ultimately, we need to perform leader election and failover reliably, and the system must rely on an authoritative system regardless of the network state. Algorithms such as Paxos guar-

antee all parties will reach the same state, and as long as the infrastructure that actually mucks with the topology relies on the consensus protocol, the correct ordering of topology changes is guaranteed, even though they may take longer than desired (distributed consensus protocols cannot guarantee when parties reach a value, just that they will, eventually). While the system may experience longer-than-desired outage times until all parties get the right value, the time (hopefully) will be less than it would be had we relied on manual operations to manage this process.

Note that in practice this is not an easy undertaking; many experienced engineers/administrators are of the opinion you are better off having manual failover and living with the increased outage times when it does happen, as this process has to be perfect or risk worse problems (for example, data corruption, incorrect topologies, and so on). Additionally, many message queues and database systems do provide guidance and some mechanism for failover that may be good enough, but most of these systems do not (and cannot) guarantee correct topologies in true network partitions and may result in "split-brain" scenarios. Therefore, if your system must support a tight availability SLA and you need strong consistency, there is not much choice in the matter.

Thankfully, we have designed the example system in such a way the actual conversion servers are stand-alone (modulo the message queue). As such, there is not anything to cover with them that is not covered elsewhere.

Platform Components

In the system as described here, only a subset of the requirements is actually supporting the core problem definition of image resize. In addition to the image requirements, the system must support authentication/authorization and usage collection (many more components would be needed to run a business, but for the purposes of this article I will cap it here). Also, to avoid repetition, this section will touch only on the highlights of why the architecture looks like it does rather than continue to step through each of the categories.

Whether you are able to leverage existing components or build your own, all of them will need to consider how to manage production deployment at scale.

Identity and authentication. The example service needs an identity-management system that supports authentication, authorization, and user management. First, the business can identify that customer management is going to occur far less often than Web service requests (if not, it is out of business). So the business hopes to serve orders-of-magnitude more authentication requests than management requests. Additionally, since this is a critical function but does not add value to the actual service, latency should be as low as possible. Given these two statements, we can see that scaling the authentication system is necessary to keep up with service requests—authentication is, generally speaking, not cacheable if done correctly.

While authentication events themselves are not cacheable, however, the data required to serve an authentication request is, and so we can still treat this system as largely read-only. This observation allows the decoupling of the "data plane" from the "management plane" and scales them independently. The trade-off is you end up with two separate systems, and in any such case, consistency between them will, by necessity, be asynchronous/eventual. In practical terms this means introducing delay between customer creation/update and the availability of this data in the data plane. This technical decision (the A in CAP) does impact the business, as new/updated customers may not be able to use the system right away. The system may need to maintain an aggressive SLA for data replication, depending on business requirements.

Usage collection. Usage collection is the direct inverse problem of authentication; it is highly write intensive, and the data is read infrequently—often only once—usually to process billing information (business analytics such as data warehousing are outside the scope of this article). Additionally, consistency of metering data is not important, as the only consumer is (typically) internal tools. Some important trade-offs can be made regarding durability; any single record or grouping of records is not valuable until aggregated into a final customer-visible accounting. You can

therefore separate durability SLAs appropriately. The difficult aspect of metering is the need to process a larger number of records than requests. The system is required to store usage data for more axes than just requests, often for a long period of time—for compliance or customer service among others—and must offer acceptable read latency for any reasonable time slice of this data. For the purposes of this example, we are describing usage record capture and aggregation, not billing or business analytics.

Given these requirements, a reasonable architectural approach would be to perform local grouping of usage records on each API server receiving customer requests, and then on time/size boundaries before sending those off to a staging system that is write-scalable and, eventually, consistent. Once records are stored there, high durability is expected, so a dynamo-flavored key/value system would be a good choice for this tier. We can periodically run aggregation batch jobs on all this data. A MapReduce system or similar distributed batch-processing system makes sense as a starting point. Data must be retained for a long period of time—up to years—so using the batch-processing system provides an opportunity to aggregate data and store raw records along customer boundaries for easier retrieval later. This data can then be heavily compressed and sent to long-term, slow storage such as (at best) low-cost redundant disk drives or tape backups.

Alternative schemes could be considered, given slightly different requirements. For example, if having real-time visibility into usage data is important, then it would be reasonable instead to use a publish/subscribe system to route batches, or even every unique usage record, to some fleet of partitioned aggregation systems that keep "live" counts. The trade-off is that such a system is more costly to scale over time as it needs to be scaled nearly linearly with API requests.

Here are a few key takeaways on architecture:

▸ Decompose the "business" application into discrete services on the boundaries of fault domains, scaling, and data workload (r/w).

▸ Make as many things as possible stateless.

▸ When dealing with state, deeply understand CAP, latency, throughput, and durability requirements.

The remainder of this article briefly addresses the rest of the life cycle of such a system.

Implementation

Now that we have walked through the initial architecture of an image-processing service—which should be viable as an initial attempt—let's touch on a few misconceptions in implementing such a system. No opinion will be offered on operating systems, languages, frameworks, and others—simply a short set of guiding principles that may be useful.

When a new engineer sets out to implement such a system as described here, the typical first approach is immediately to survey available software—usually open source—and "glue" it together. Hopefully after taking the whirlwind tour of architecting such a system, you will recognize this approach as unwise. While it may work at development scale, every system is implemented with a set of assumptions by the engineer(s) who wrote it. These assumptions may or may not align with the requirements of your distributed system, but the point is, it is your responsibility to understand the assumptions that a technology makes, and its trade-offs, and then assess whether this software works in your environment. The common case is that some software will align with your assumptions, and some will not.

For the situations where you need to develop your own software to solve your system's problems, the next step should be assessing prior art in the space. Solutions can usually be found in some existing technology—often from longer ago than one would expect—but the assumptions of the system that implements this solution are different from yours. In such cases, break the system's strong points down into first principles, and "rebuild" on those. For example, in the authentication system described here, an existing Lightweight Directory Access Protocol (LDAP) solution would likely not be the right choice, given the assumptions of the protocol and exist-

ing LDAP system replication models, but we could certainly reuse many of the ideas of information management and authentication mechanisms, and implement those basic primitives in a way that scales for our environment.

As another example, if we need to implement or use a storage solution, many off-the-shelf solutions to this day involve some type of distributed file system (DFS) or storage area network (SAN), whether in the form of NFS (file), iSCSI (block), or some other protocol. But as described earlier, these systems all choose C in the CAP theorem, and in addition are often unable to scale past the capacity of a single node. Instead, the basic primitives necessary for many applications are present in modern object stores, but with different trade-offs (for example, no partial updates) that allow these storage systems to handle distributed scaling of some applications better.

Whether you are able to leverage existing components or build your own, all of them will need to consider how to manage production deployments at scale. I will not delve into the details here but point out only that configuration-management solutions should be looked at, and a system that can effectively manage deployments/rollback and state tracking of many systems is needed.

Testing and validation of a distributed system is extremely difficult—often as or more difficult than creating the system in the first place. This is because a distributed system simply has so many variables it is nearly impossible to control them all in isolation; therefore, for the most part, a system must be empirically validated rather than being proved theoretically correct. In particular, total outages in networking are relatively uncommon, but degraded service for brief periods of time is extremely likely. Such degradation often causes software to act in different ways from those encountered during development, resulting in the system software being the most likely cause of failure. Thus, the number-one cause of failures in distributed systems is without doubt heisenbugs—transient bugs that are often repeatable but diificult to reproduce, and dependent on some sequence of events in the system taking place.[7]

Validation of a distributed system involves many aspects, and certainly enough load needs to be driven through the system to understand where it "tips over" such that intelligent capacity planning and monitoring can be done. Even more so, though, failures and degraded states must be introduced regularly; failures can and will happen, and every component's dependencies must be taken away from it to understand the system acts as appropriate (even if that means returning errors to the customer) and the system recovers when the dependencies recover. The Netflix Chaos Monkey[2] is one of the best public references to such a methodology; automated failure/degradation to its test and production systems is considered essential to running its business.

Operations. Last, but certainly not least, are the operations of a distributed system. In order to run a system as described in this article, we must be able to capture and analyze all aspects of the hardware and operating systems, network(s), and application software—in short, as the mantra goes, "if it moves, measure it." This has two main aspects: realtime monitoring to respond to failures; and analytics—both realtime and historical—allowing operators, engineers, and analysts to understand the system and trends.

As may be obvious by this point, the amount of data a large distributed system can generate for operational data may, and likely will, exceed the amount of data that actually serves the business function. Storage and processing of this scale of data will likely require its own distributed system that can "keep up," which essentially brings us back to the beginning of this article: you'll need a distributed system to operate and manage your distributed system.

PaaS and Application Outsourcing
A final consideration is the recent adoption of Platform as a Service (PaaS), such as Google App Engine[6] or Heroku.[8] Such offerings have grown very popular as they reduce the engineering effort of delivering a vertical application to market, such as a traditional MVC (model-view-controller) Web application. By now, it should be obvious there are no magic bullets, and it is imprudent simply to hand over your system to a third party without due diligence. PaaS systems will indeed work for many applications—quite well, in fact—but again, only if your system requirements align with the feature set and trade-offs made by the PaaS provider; the former is fairly easy to evaluate, and the latter may be difficult or impossible. As an example, reference the recent public disclosure by Rap Genius[9] regarding Heroku's routing trade-offs. Ignoring all legal aspects, the problem encountered was ultimately a severe degradation of application latency caused by implementation trade-offs that suited the platform's requirements rather than the tenant's. It may well be the case the business was still correct in choosing PaaS, but the point is this issue was incredibly difficult to find a root cause, and the business hit it at the worst possible time: under production load as the application scaled up.

Conclusion
Distributed systems are notoriously difficult to implement and operate. Nonetheless, more engineers are finding themselves creating distributed applications as they move into running modern-day Web or mobile applications, or simply move an existing enterprise application onto a cloud service provider.

Without practical experience working on successful—and failed—systems, most engineers take a "hopefully it works" approach and attempt to string together off-the-shelf software, whether open source or commercial, and often are unsuccessful at building a resilient, performant system. In reality, building a distributed system requires a methodical approach to requirements along the boundaries of failure domains, latency, throughput, durability, consistency, and desired SLAs for the business application at all aspects of the application. Ⓒ

Ⓠ **Related articles on queue.acm.org**

Distributed Computing Economics
Jim Gray
http://queue.acm.org/detail.cfm?id=1394131

Monitoring and Control of Large Systems with MonALISA
Iosif Legrand, Ramiro Voicu, Catalin Cirstoiu, Costin Grigoras, Latchezar Betev, Alexandru Costan
http://queue.acm.org/detail.cfm?id=1577839

Condos and Clouds
Pat Helland
http://queue.acm.org/detail.cfm?id=2398392

References
1. Allspaw, J. *The Art of Capacity Planning.* O'Reilly Media, 2008; http://shop.oreilly.com/product/9780596518585.do.
2. Bennett, C. and Tseitlin, A. Netflix: Chaos Monkey released into the wild. Netflix Tech Blog; http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html.
3. Chandra, T., Griesemer, R. and Redstone, J. 2007. Paxos made live. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing* (2007), 398–407; http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/archive/paxos_made_live.pdf.
4. Dean, J. and Ghemawat, S. MapReduce: simplified data processing on large clusters. In *Proceedings of 6th Symposium on Operating Systems Design and Implementation I* (2004); http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/archive/mapreduce-osdi04.pdf.
5. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P, and Vogels, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating System Principles* (2007); http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf.
6. Google App Engine. Google Developers; https://developers.google.com/appengine/.
7. Gray, J. Why do computers stop and what can be done about it. Tandem Technical Report 85.7, 1985; http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.59.6561.
8. Heroku Cloud Application Platform; http://www.heroku.com/.
9. Somers, J. Heroku's ugly secret; http://rapgenius.com/James-somers-herokus-ugly-secret-lyrics.
10. Tarreau, W. Make applications scalable with load balancing; http://1wt.eu/articles/2006_lb/.

Mark Cavage is a software engineer at Joyent, where he works primarily on distributed-systems software and maintains several open source toolkits, such as ldapjs and restify. He was previously a senior software engineer with Amazon Web Services, where he led the Identity and Access Management team.